1 Introduction

In this work we provide some initial structural complexity results for classes of probabilistically checkable proof systems (PCPs) for nondeterministic NC (NNC) circuit families, as a first step towards characterizing probabilistic proof systems for P. One of the major successes of the characterization of NP as a class of PCPs (with polynomial-time verifiers) is that it provides a route to proving that approximating certain computationally intractable optimization problems is as difficult as solving them exactly. The growth of multiprocessor systems in general purpose personal computers highlights the urgency of proving the analogous inapproximability (or approximability) of inherently sequential optimization problems by highly parallel algorithms. The P-complete problems are generally considered to be inherently sequential, whereas problems in NC are considered highly parallel. Our guiding question is now, “For which P-complete problems are there NC approximation algorithms, and for which are there none?”

Unfortunately, the techniques used to prove the original PCP Theorem rely on the fact that NP can be interpreted as the class of languages for which there is an efficient verification procedure given a brief witness to language membership; no such obvious interpretation of P exists. Indeed, this question was already on the minds of researchers soon after the original proof of the PCP Theorem.

An intriguing question is whether the known non-approximability results for sequential algorithms can be improved when we restrict to NC algorithms (under the assumption that P ≠ NC). A possible way may be to devise probabilistic proof systems for P more efficient that the currently known proof systems for NP. Such a result would have a great independent interest. However, it is not clear why proofs for
P should be easier to check than proofs for NP (they only appear to be easier to generate). [18]

Perhaps P has proof systems which are easy to check in NC, but this remains unclear. Instead, we consider proof systems for the class NNC(polylog), the class of languages decidable by NC circuit families augmented with a polylogarithmic number of nondeterministic gates. We hope that this will be a valuable first step toward understanding proof systems for P. We consider NNC(polylog) for two reasons. First, it is defined in such a way that it explicitly has short proof systems which are easy to verify efficiently. Second, it, like P, lies between NC and NP.

Although our original intention was to show a result like NNC(polylog) = PCP[O(log log n), O(1)], our research reveals that proving such an equality is equivalent to proving NNC(polylog) = NC, or in other words, that a polylogarithmic amount of nondeterminism can be simulated deterministically by an NC circuit family. This should be seen as evidence that such a result is unlikely; in fact, we show that such a simulation implies a deterministic subexponential time algorithm for SATISFIABILITY! We are still, however, able to show that certain PCP classes are contained in NNC(polylog).

Recent work by other authors has focused on the practicality of implementing proof systems for NP on real computers. Unlike this work, in which we completely ignore the resources required for the prover to transform the proof of membership in a language to a probabilistically checkable proof in the appropriate format, other works focus on efficient implementation of both the verifier and the prover. Also, some other works consider a model of proof system in which the prover and the verifier have some limited communication. See [4, 15, 16, 17, 11] for more information.

2 Preliminaries

Throughout this work, log n denotes the base 2 logarithm of n. In the definitions below, \( \mathbb{N} \) denotes the set of non-negative integers and \( \mathbb{R} \) denotes the set of real numbers.

**Definition 2.1.** For all functions \( f, g : \mathbb{N} \to \mathbb{R} \), the function \( f \) is in the class \( O(g(n)) \) if there exist real numbers \( c \) and \( N \) such that for all natural numbers \( n \) we have \( n > N \) implies \( f(n) \leq c \cdot g(n) \). If \( f(n) < c \cdot g(n) \) then \( f(n) \) is in \( o(g(n)) \). If \( f(n) \geq c \cdot g(n) \) then \( f(n) \) is in \( \Omega(g(n)) \). If \( f(n) > c \cdot g(n) \) then \( f(n) \) is in \( \omega(g(n)) \).

We assume the reader knows the basic definitions from complexity theory, including those of the complexity classes P, NP, DTIME, and DSPACE. We define the class \( L^k \) by \( L^k = \text{DSPACE}(\log^k n) \) for all non-negative integers \( k \) and the class \( \text{polyL} \) by \( \text{polyL} = \bigcup_{k \in \mathbb{N}} \text{DSPACE}(\log^k n) \). We denote the class \( L^1 \) by simply L. We define the complexity class SUBEXP, the class of languages decidable by
deterministic “subexponential” time Turing machines, as
\[ \text{SUBEXP} = \cap_{\epsilon > 0} \text{DTIME}(2^{n^\epsilon}) \]
and QP, the class of languages decidable by a deterministic “quasipolynomial”
time Turing machine, as
\[ \text{QP} = \bigcup_{k \in \mathbb{N}} \text{DTIME}(2^{\log^k n}). \]
We will also be considering \( \text{NC}^k \), the class of languages decidable by a family
of logarithmic space uniform Boolean circuits of polynomial size, \( O(\log^k n) \)
depth, and unbounded fan-in. We will denote by \( \text{NC} \) the union of all the \( \text{NC}^k \)
classes. A language in \( \text{NC}^k \) can also be described as a language which admits an
algorithm which uses a polynomial number of processors running in \( O(\log^k n) \) 
time on a parallel random-access machine (PRAM). In this work, we describe
\( \text{NC} \) algorithms using this paradigm.

**Definition 2.2** ([8, Definition 1.1]). Let \( V \) be a finite set of variables, defined
by \( V = \{x_1, x_2, \ldots, x_n\} \), let \( \Gamma \) be a finite alphabet, and let \( q \) be a natural
number. A \( q \)-ary constraint is a \( q+1 \) tuple, \((C, i_1, i_2, \ldots, i_q)\), where \( C \subseteq \Gamma^q \)
and \( i_k \in \{1, 2, \ldots, n\} \) for all \( k \in \{1, 2, \ldots, q\} \). Here, \( C \) is considered the set of
“acceptable” values for the variables.

An assignment is a function \( a : V \rightarrow \Gamma \). An assignment satisfies a constraint
if \((a(v_{i_1}), a(v_{i_2}), \ldots, a(v_{i_q})) \in C\).

**Definition 2.3** (\( q \)-CSP).

**Instance:** finite alphabet \( \Gamma \) with \( |{\Gamma}| > 1 \), finite set of variables \( V \), finite set
of \( q \)-ary constraints \( D \).

**Question:** Are all constraints in \( D \) satisfiable?

**Definition 2.4** (\( \frac{1}{2} \)-GAP \( q \)-CSP).

**Instance:** finite alphabet \( \Gamma \) with \( |{\Gamma}| > 1 \), finite set of variables \( V \), finite set
of \( q \)-ary constraints \( D \) with the restriction that for any assignment, either all
constraints are satisfied or fewer than half are.

**Question:** Are all constraints in \( D \) satisfiable?

In the special case in which \( q = 2 \), that is, all constraints are binary, we may
interpret an instance of the constraint satisfaction problem as an undirected
digraph with vertex set \( V \) and an edge labeled \( C \) between vertices \( v_i \) and \( v_j \) for
each \((C, i, j)\) in \( D \). We call such a graph a constraint graph and we consider the
size of this graph to be \(|V| + |E| \) where \( V \) is the set of vertices (equivalently,
variables) and \( E \) is the set of edges.

**Definition 2.5.** A PCP verifier is a probabilistic Turing machine which has
sequential access to an input string \( x \), sequential access to a random string \( \rho \),
and nonadaptive random access to a proof string \( \pi \).

The initialism “PCP” stands for “probabilistically checkable proof”.

3
Definition 2.6. Let $r(n)$ and $q(n)$ be bounded above by polynomials in $n$, and let $c(n)$ and $s(n)$ be functions whose values are in the interval $[0, 1]$. A language $L$ has a $(r(n), q(n), c(n), s(n))$-PCP verifier if there exists a PCP verifier $V$ such that $V$ uses at most $r(n)$ bits of the random string $\rho$, makes at most $q(n)$ nonadaptive queries to bits of the proof $\pi$, and satisfies the following conditions.

1. If $x \in L$, then
   \[ \exists \pi \in \Sigma^* : \Pr_{\rho \in \Sigma^{r(n)}}[V(x, \pi; \rho) \text{ accepts}] \geq c(n). \]

2. If $x \notin L$, then
   \[ \forall \pi \in \Sigma^* : \Pr_{\rho \in \Sigma^{r(n)}}[V(x, \pi; \rho) \text{ accepts}] < s(n). \]

The value $c(n)$ is the completeness and the value $s(n)$ the soundness of the verifier.

In this work, we will consider only nonadaptive PCP verifiers. Since a (nonadaptive) $(r(n), q(n), c(n), s(n))$-PCP verifier can read at most $2^{r(n)}q(n)$ locations of the proof string with nonzero probability, we assume without loss of generality that the proof provided to the verifier is of length at most $2^{r(n)}q(n)$ [2, Remark 11.6]. (Note that a verifier which uses $q(n)$ adaptive random access queries to the proof string can be simulated by a verifier which uses $2^{q(n)}$ nonadaptive random access queries to the proof string, so in the adaptive case, the proof string could be of length $2^{r(n)}+q(n)$.)

Definition 2.7. Let $\text{PCP}^{c(n), s(n)}_{r(n), q(n)}[\mathcal{F}, \mathcal{G}]$ be the class of all languages $L$ such that $L$ has a $(r(n), q(n), c(n), s(n))$-PCP verifier $V$ computable by a $\mathcal{C}$ algorithm.

More generally, if $\mathcal{F}$ and $\mathcal{G}$ are classes of functions,

\[
\text{PCP}^{c(n), s(n)}_{r(n), q(n)}[\mathcal{F}, \mathcal{G}] = \bigcup_{f \in \mathcal{F}, g \in \mathcal{G}} \text{PCP}^{c(n), s(n)}_{r(n), q(n)}[f(n), g(n)].
\]

Since completeness 1 and soundness $\frac{1}{2}$ are common parameters, and for the sake of brevity, we write $\text{PCP}^{c}[r(n), q(n)]$ to denote $\text{PCP}^{c}_{1, \frac{1}{2}}[r(n), q(n)]$, and $\text{PCP}^{c}[\mathcal{F}, \mathcal{G}]$ to denote $\text{PCP}^{c}_{1, \frac{1}{2}}[\mathcal{F}, \mathcal{G}]$.

Please notice that the complexity class given in the superscript in the above definition does not denote an oracle; it merely describes the computational power of the PCP verifier.

From the definition, we see immediately that

\[
\text{PCP}^{c(n), s(n)}_{r(n), q(n)}[O(r(n)), O(q(n))] = \bigcup_{a \in \mathbb{N}, b \in \mathbb{N}} \text{PCP}^{c(n), s(n)}_{r(n), q(n)}[a \cdot r(n), b \cdot q(n)].
\]
3 Probabilistically checkable proofs for non-deterministic circuits

We first provide a PCP characterization of NNC(polylog), then later we provide upper and lower bounds for the randomness and query complexity parameters of such a PCP verifier.

**Theorem 3.1.** For all non-negative integers q and r,

\[
\text{NNC}(\log^q n) \subseteq \text{PCP}^{\text{NC}}[r \log \log n, O(\log^q n)] \subseteq \text{NNC}(\log^{q+r} n),
\]

and specifically

\[
\text{NNC}(\log^q n) \subseteq \text{PCP}^{\text{NC}}[\log \log n, O(\log^q n)] \subseteq \text{NNC}(\log^{q+1} n).
\]

**Proof.** Let q and r be non-negative integers. The inclusion \(\text{NNC}(\log^q n) \subseteq \text{PCP}^{\text{NC}}[r \log \log n, O(\log^q n)]\) follows immediately from the definitions. For the other direction, suppose \(L \in \text{PCP}^{\text{NC}}[r \log \log n, c \log^q n]\) for some constant \(c\).

1. Guess a proof string \(\pi\) of length \(2r \log \log n\).

2. For each \(\rho\) of length \(r \log \log n\) in parallel simulate \(V(x, \pi; \rho)\).

3. Accept if and only if at least half of the simulations accept.

In the initial step, guessing a proof string requires \(O(\log^{q+r} n)\) bits. In the second step, since \(V\) is an NC machine, a polylogarithmic number of parallel simulations of \(V\) can be executed with only a polylogarithmic factor increase in size and no increase in depth. In the final step, computing the majority of a polylogarithmic number of bits can be done by an NC circuit. Therefore \(M\) is an NNC(\(\log^{q+1} n\)) machine. The correctness of \(M\) follows from the completeness and soundness of \(V\).

**Corollary 3.2.** \(\text{NNC}(\text{polylog}) = \text{PCP}^{\text{NC}}[O(\log \log n), \text{polylog}]\).

Next, consider the chain of inclusions

\[
\text{NNC}(\log^q n) \subseteq \text{NNC}(\text{polylog}) \subseteq \text{NNC}(\text{poly}).
\]

In fact, \(\text{NC} = \text{NNC}(\log n)\) and \(\text{NNC}(\text{poly}) = \text{NP}^{[10]}\), so we can rewrite this as

\[
\text{NC} \subseteq \text{NNC}(\text{polylog}) \subseteq \text{NP}.
\]

We now wish to provide NC PCP characterizations for both NC and NP.

In [10], the authors prove that \(P = \text{PCP}^P[O(\log \log n), O(1)]\) (implicitly; they state only that \(\text{NP} = \text{PCP}^P[O(\log \log n), O(1)]\) if and only if \(P = \text{NP}\)). The same proof techniques can be used in the NC setting with essentially no changes. (The idea of the proof is to simulate \(O(\log \log n)\) bits of randomness with \(\log \log n + O(1)\) bits by making a random walk of an appropriate length on a fully explicit constant degree expander graph.)
Theorem 3.3. NC = PCP^{NC}[O(log log n), O(1)].

As a generalization of the result of [10], we know \( \text{NP} = \text{PCP}^{P}[o(log n), o(log n)] \) if and only if \( P = \text{NP} \) \([1, 9]\). Unfortunately, the obvious strategy for translating that proof to the NC setting fails. The proof would have shown that \( \text{NP} = \text{PCP}^{NC}[o(log log n), O(1)] \) if and only if \( \text{NC} = \text{NP} \), but this is already proven by Theorem 3.3 and the fact that \( \frac{\log log n}{\log n} \leq \log n \).

We also know the following strengthening of the original PCP Theorem; the proof of this theorem is in section 5.

Theorem 3.4. PCP^{NC}[O(log n), O(1)] = \text{NP}.

From Equation 1, Theorem 3.3, and Theorem 3.4, we have the two equivalent inclusion chains

\[
\text{NC} \subseteq \text{NNC}(\text{polylog}) \subseteq \text{NP}
\]

and

\[
\text{PCP}[O(log log n), O(1)] \subseteq \text{PCP}[O(log log n), \text{polylog}] \subseteq \text{PCP}[O(log n), O(1)],
\]

where the PCP verifier is an NC machine. If we can provide evidence that \( \text{NC} \neq \text{NNC}(\text{polylog}) \) and that \( \text{NNC}(\text{polylog}) \neq \text{NP} \), we can conclude that the corresponding PCP classes are also likely distinct. This theorem, adapted from [7, Theorem 1] (therein attributed to R. Beigel), provides that evidence.

\textbf{TODO} There is further evidence that \( \text{NC} \neq \text{NNC}(\text{polylog}) \): the fact that \( \text{NNC}(\text{polylog}) = \text{NC} \subseteq \text{P} \) implies \( W[P] \) collapses to FPT

Theorem 3.5.

1. If \( \text{NC} = \text{NNC}(\text{polylog}) \), then \( \text{NP} \subseteq \text{SUBEXP} \).

2. If \( \text{NNC}(\text{polylog}) = \text{NP} \), then \( \text{NP} \subseteq \text{QP} \).

Each of the two conclusions in this theorem implies that the exponential time hypothesis is false. Furthermore, in the latter case, the conclusion implies that \( \text{EXP} = \text{NEXP} \).

Proof of Theorem 3.5. If \( \text{NNC}(\text{polylog}) = \text{NP} \), then

\[
\text{NP} = \text{NNC}(\text{polylog}) \\
\subseteq \text{DSPAC}(\text{polylog}) \quad \text{by} \; [19] \\
\subseteq \text{DTIME}(2^{\text{polylog}}) \quad \text{by exhaustive search} \\
= \text{QP} \quad \text{by definition}.
\]

Now suppose \( \text{NC} = \text{NNC}(\text{polylog}) \). Since \text{SATISFIABILITY} is complete for \text{NP} under deterministic polynomial-time many-one reductions, it suffices to show a deterministic subexponential time algorithm for \text{SATISFIABILITY}.

The proof uses a padding argument. First we observe that there is an \( \text{NNC}^{1}(n) \) machine, call it \( M \), that decides \text{SATISFIABILITY}: given a Boolean
The correctness of this algorithm follows immediately from the correctness of the formula \( \phi \). Let \( \epsilon \) be an arbitrarily small positive constant, and define \( L \), the padded version of Satisfiability, as

\[
L = \left\{ \phi \#^1 P \mid \phi \in \text{Satisfiability and } P = 2^{n^\epsilon} - (n + 1) \right\},
\]

where \( n = |\phi| \). We claim \( L \) is in \( \text{NNC}(\log^2 n) \) by the following machine, \( M_j \). On input \( \phi' \), check that \( \phi' \) is in the format \( \phi \#^1 P \), then accept if and only if \( M \) accepts \( \phi \). The correctness of this algorithm follows from the correctness of \( L \), so it remains to check the size and depth of the circuit for \( M_j \), and the amount of nondeterminism used.

Checking that \( x' \) is in the correct format can be performed (deterministically) by an \( \text{NC}^1 \) circuit by computing the conjunction of all the bits after the \# symbol. Observe now that \( |x'| = 2^{n^\epsilon} \), so \( n = \log^2 |x'| \). The amount of nondeterminism used by \( M_j \) is the same as the amount used by \( M \), which is \( O(n) \), or \( O(\log^2 |x'|) \).

The size of \( M \) is polynomial in \( n \), which is polylogarithmic in \( |x'| \), and hence polynomial in the length of the input \( x' \). The depth of \( M \) is \( O(\log n) \), which is \( O(\log \log^2 |x'|) \), or simply \( O(\log \log |x'|) \). We conclude that the size of \( M_j \) is polynomial in \( |x'| \), the depth of \( M_j \) is logarithmic in \( |x'| \), and \( M_j \) uses \( O(\log^{1/2} |x'|) \) bits of nondeterminism. Hence \( L \) is in \( \text{NNC}(\log^{3/2} n) \).

By hypothesis, \( L \) is also in \( \text{NC} \). Let \( M_i \) be the \( \text{NC} \) machine that decides it. We claim that we can now construct a subexponential time algorithm for Satisfiability on inputs \( \phi \) of length \( n \).

1. Let \( \phi' = \phi \#^1 P \), where \( P = 2^{n^\epsilon} - (n + 1) \).
2. Accept if and only if \( M_i \) accepts \( \phi' \).

The correctness of this algorithm follows immediately from the correctness of \( M_i \). The first step can be performed by a deterministic algorithm running in time \( 2^{n^\epsilon} \). The second step can be performed by an \( \text{NC} \) machine. Since \( \text{NC} \subseteq \text{P} \), and \( 2^{n^\epsilon} \) is greater than any polynomial for sufficiently large \( n \), the first step is the bottleneck in this algorithm. Therefore, this algorithm for Satisfiability can be implemented by a deterministic algorithm running in \( O(2^{n^\epsilon}) \) time for arbitrarily small \( \epsilon \).

One interpretation of this theorem is that the difference in complexity between languages in \( \text{NC} \) and languages in \( \text{NNC}(\text{polylog}) \) is smaller than the difference in complexity between languages in \( \text{NNC}(\text{polylog}) \) and languages in \( \text{NP} \). This makes sense: the ratio of a polynomial in \( \log n \) to \( \log n \) is smaller than the ratio of a polynomial in \( n \) to a polynomial in \( \log n \).

**Corollary 3.6.**

1. If \( \text{PCP}^{\text{NC}}[O(\log \log n), O(1)] = \text{PCP}^{\text{NC}}[O(\log \log n), \text{polylog}] \), then \( \text{NP} \subseteq \text{SUBEXP} \).
2. If \( \text{PCP}^{\text{NC}}[O(\log \log n), \text{polylog}] = \text{PCP}^{\text{NC}}[O(\log n), O(1)] \), then \( \text{NP} \subseteq \text{QP} \).
The first part of this corollary provides evidence that for certain classes of computational problems, an NC PCP verifier cannot reduce the number of necessary queries. (However, it could still be the case that for some fixed positive integer $k$, we have $\text{PCP}^{\text{NC}}[O(\log \log n), O(1)] = \text{PCP}^{\text{NC}}[O(\log \log n), O(\log^k n)]$; see Conjecture 3.7 below.) The second part provides evidence that for certain classes of computational problems, a verifier cannot reduce randomness in exchange for an increase in the number of necessary queries. Contrast this with [10, Corollary 10] which states that $\text{PCP}^{\text{NC}}[O(\log \alpha \log n), O(\log \log n)] \subseteq \text{PCP}^{\text{NC}}[O(\log \log n), O(\log \log n)]$, which provides an even more severe collapse, assuming the following conjecture.

**Conjecture 3.7.** $\text{PCP}^{\text{NC}}[O(\log \log n), O(\log n)] = \text{PCP}^{\text{NC}}[O(\log \log n), O(1)]$.

This is a scaled down version (that is, scaled from a $P$ verifier down to an NC verifier) of some of the results of the research which led to the original PCP Theorem. If this conjecture holds, then $\text{poly}(\log \log n)$ randomness and a logarithmic number of queries to the proof can be simulated deterministically.

**Theorem 3.8.** If Conjecture 3.7 holds, then $\text{PCP}^{\text{NC}}[\text{poly}(\log \log n), O(\log n)] = \text{NC}$.

**Proof.** Combining Conjecture 3.7 with the fact that $O(\log^\alpha \log n) \subseteq O(\log n)$ for all non-negative integers $\alpha$, we have

\[
\text{NC} \subseteq \text{PCP}^{\text{NC}}[\text{poly}(\log \log n), \text{poly}(\log \log n)] \\
\subseteq \text{PCP}^{\text{NC}}[O(\log \log n), \text{poly}(\log \log n)] \\
\subseteq \text{PCP}^{\text{NC}}[O(\log \log n), O(\log n)] \\
\subseteq \text{PCP}^{\text{NC}}[O(\log \log n), O(1)] \\
\subseteq \text{PCP}^{\text{NC}}[O(\log \log n), O(1)] = \text{NC}.
\]

Now we return to our original goal, finding a PCP characterization of $P$. The classes $P$ and $\text{NNC}(\text{polylog})$ are conjectured incomparable [19]. Using the results above, this conjecture implies that $P$ and $\text{PCP}^{\text{NC}}[O(\log \log n), \text{polylog}]$ are incomparable. Theorem 3.9 shows negative consequences of a PCP characterization for $P$.

**Theorem 3.9.**

1. If $P \subseteq \text{PCP}^{\text{NC}}[O(\log \log n), \text{polylog}]$ then $P \subseteq \text{polyL}$.
2. If $\text{polyL} \subseteq P$ then $\text{PCP}^{\text{NC}}[O(\log \log n), \text{polylog}] \not\subseteq P$. 
Proof. These implications are a consequence of three facts.

1. $\text{PCP}^{\text{NC}}[O(\log \log n), \text{polylog}] \subseteq \text{NNC}(\text{polylog})$ (Corollary 3.2).

2. $\text{NNC}(\text{polylog}) \subseteq \text{polyL}$ ([19, Corollary 3.2]).

3. $P \neq \text{polyL}$ ([5, Theorem 3.10]).

Although $P \neq \text{polyL}$, whether one is a strict subset of the other remains unknown; the two are conjectured to be incomparable [12, Section 2.5.1]. If $P \subseteq \text{polyL}$, then $P \subseteq \text{QP}$ (by exhaustive search over the quasipolynomial number of configurations of the polyL machine). If $\text{polyL} \subseteq P$, then $L \subseteq L^2 \subseteq \cdots \subseteq \text{polyL} \subseteq P$.

4 Inapproximability from PCPs

Consider the maximum high degree subgraph problem.

**Definition 4.1 (Maximum High Degree Subgraph).**

**Instance:** undirected graph $G$, integer $d$.

**Solution:** subset $S$ of vertices whose induced subgraph has minimum degree at least $d$.

**Measure:** $|S|$.

**Type:** maximization.

This problem is a relaxation of the maximum clique problem, in which we require that the minimum degree of the induced subgraph is at least $|S| - 1$. There is a simple polynomial-time algorithm that outputs optimal solutions for this problem: repeatedly remove vertices of degree less than $d$ from the graph.

We know that the clique problem is inapproximable via a gap-introducing reduction from an arbitrary probabilistically checkable proof system [9]. Can we use a similar reduction to provide a gap-introducing reduction from our restricted PCPs to the maximum high degree subgraph problem?

5 Probabilistically checkable proofs for nonde-terministic polynomial time

One method of showing $\text{PCP}^{\text{NC}}[O(\log n), O(1)] = \text{NP}$ is to revisit a proof of the PCP Theorem and ensure that all computation can be performed by an NC PCP verifier without affecting the correctness of the proof. We will consider Dinur’s proof of the PCP Theorem [8], which reduces the problem of proving $\text{PCP}^{\text{P}}[O(\log n), O(1)] = \text{NP}$ to the problem of showing $1/3$-gap $q$-CSP is hard for $\text{NP}$. Meir observes that although we would like a verifier running in polylogarithmic time, Dinur’s proof requires $O(\log n)$ iterations of a polynomial-time procedure, which yields a polynomial-time procedure [13, Section 1.2.1]. We show that a closer look reveals that parallel polylogarithmic time is indeed possible without any new machinery.
We begin with the $\text{NC}$ version of the necessary reduction.

**Lemma 5.1.** If there is a positive integer $q$ such that $1/2$-GAP $q$-CSP is hard for $\text{NP}$ under $\text{NC}$ many-one reductions, then $\text{PCP}^{\text{NC}}[O(\log n), O(1)] = \text{NP}$.

**Proof.** One inclusion in the conclusion of the theorem is true unconditionally, following from the PCP Theorem [3]. For the other inclusion, let $L$ be a language in $\text{NP}$. By hypothesis there is a many-one reduction computable in $\text{NC}$ from $L$ to $1/2$-GAP $q$-CSP. We construct the PCP verifier as follows.

1. Compute the reduction to produce a set of constraints.
2. Use $O(\log n)$ random bits to choose a constraint uniformly at random.
3. Check that the constraint is satisfied by querying the proof string at the appropriate locations (the locations corresponding to the $q$ variables in the constraint).

If $x \in L$ then all constraints are satisfiable, so there exists an assignment such that the verifier will accept on all random choices of the constraint. If $x \notin L$ then fewer than half of the constraints are satisfiable, so for any assignment the probability that the verifier will select a satisfied constraint is less than half. Therefore we have shown a correct PCP verifier with the appropriate parameters for an arbitrary language in $\text{NP}$. $\square$

Now we examine Dinur’s proof that $1/2$-GAP $q$-CSP is hard for $\text{NP}$ [8]. That proof shows that the problem is hard under polynomial-time many-one reductions, but we show here that it is in fact hard under $\text{NC}$ many-one reductions. First, we claim without proof that $q$-CSP is hard for $\text{NP}$ under $\text{NC}$ many-one reductions (because the standard polynomial-time many-one reductions showing that it is $\text{NP}$-complete are in fact computable in logarithmic space). Next, consider (the high-level description of) the polynomial-time many-one reduction from $q$-CSP to $1/2$-GAP $q$-CSP: given constraint graph $G_0$ as input, compute and output $G_{O(\log n)}$, where $G_{i+1} = \mathcal{P}(\mathcal{X}(G_i)^t)$ and $t \in O(1)$. Here, $\mathcal{X}$ is a preprocessing function, the exponent $t$ denotes a constant number of constraint graph powering operations, and $\mathcal{P}$ denotes an assignment testing composition function. If each of these three functions is computable by an $\text{NC}$ algorithm, then $G_{t+1}$ can be computed from $G_t$ by an $\text{NC}$ algorithm, and hence so can $G_{O(\log n)}$ from $G_0$. We will consider each of the three functions below.

The preprocessing function $\mathcal{X}$ requires a *mildly explicit* construction of a constant degree expander. The standard definition of mildly explicit is that a representation of the graph (for example, its adjacency matrix) is computable in time polynomial in the number of vertices in the graph; for comparison, in a *fully explicit* expander, the $i$th neighbor of vertex $v$ can be computed in time polynomial in the size of the binary representation of $v$, that is, polynomial in $\log n$ where $n$ is the number of nodes in the graph. We will refer to graphs which meet these definitions as *polynomial-time mildly explicit* and *polynomial-time fully explicit*. Since we are constructing an $\text{NC}$ algorithm, we will require the
representation of the graph to be computable by an NC algorithm. More formally, we require an NC mildly explicit graph, that is, a graph for which a representation can be computed by an NC algorithm with respect to input $n$, the number of nodes of the graph. Fortunately, an NC mildly explicit expander is equivalent to a polynomial-time fully explicit expander.

**Proposition 5.2.** Suppose $G$ is a $d$-regular expander graph. $G$ is NC mildly explicit if and only if $G$ is polynomial-time fully explicit.

**Proof.** First suppose $G$ is NC mildly explicit, so there exists an NC algorithm that outputs, say, the adjacency list of the graph given $n$ as input. The following algorithm computes the $i$th neighbor of vertex $v$ in time polynomial in $\log n$: construct the adjacency list of the graph, then find and output the $i$th neighbor of vertex $v$. When constructing the adjacency list, we simulate each of the polynomial number of processors sequentially, each one running for $\text{poly}(\log n)$ time, so the overall running time of this step remains $\text{poly}(\log n)$. The list corresponding to vertex $v$ can be found in $O(\log n)$ steps (by using a binary search tree), and the $i$th element of that list can be found in constant time (since $i$ is bounded above by $d$, a constant). Therefore we have presented an algorithm which runs in $\text{poly}(\log n)$ time which correctly computes the $i$th number of $v$ in the graph $G$.

Now suppose $G$ is polynomial-time fully explicit, so there exists an algorithm which computes the $i$th neighbor of $v$ in time polynomial in $\log n$. Let $f(v, i)$ denote this algorithm. The following NC algorithm computes the adjacency list of $G$ given the number of nodes $n$: for each vertex $v$ in parallel and each index $i$ less than $d$ in parallel add $f(v, i)$ to the list corresponding to $v$. This algorithm can be computed with $dn$ processors, which is polynomial in $n$. Since the $f(v, i)$ can be computed in time polynomial in $\log n$, the running time for each parallel processor is also polynomial in $\log n$. Therefore we have presented an NC algorithm which correctly computes a representation of the graph $G$. \qed

This proposition allows us to replace any polynomial-time mildly explicit expander in Dinur’s proof with a polynomial-time fully explicit one. (polynomial-time fully explicit constant degree expander graphs exist; see [14], for example.)

Now, let us return to the preprocessing function $X$, which is defined in two parts, [8 Definition 4.1] and [8 Definition 4.2]. In the first part, each vertex $v$ is replaced by an NC mildly explicit $d$-regular expander on $\deg(v)$ vertices in which the constraints on the edges of the expander are the equality constraint. In the second part, a constant number of self-loops along with the edges of a $d'$-regular NC mildly explicit expander on $n$ vertices are added to the graph with null constraints on the added edges. Both of these parts are computable by an NC algorithm; note that the size of the output graph in each case is linear in the size of the input graph, so a linear number of processors will suffice (with an additional multiplicative factor of a polynomial number of processors when constructing the expander graphs). We conclude the following.

**Lemma 5.3.** The preprocessing function $X$ is computable by an NC algorithm.
Constraint graph powering, defined in [8] Section 1.2, is the standard graph powering operation with an additional operation on the alphabet and the set of constraints. Standard graph powering can be performed in NC because matrix multiplication is in NC, and a constant number of matrix multiplications remains in NC. If the alphabet is of constant size, the power graph also has an alphabet of constant size, and each of the constraints becomes a new constraint of constant size (see [8] Section 1.2 for the constraint construction). Each of the constraints (which is the same as the number of edges) can be written by a distinct processor in parallel constant time. We conclude the following.

**Lemma 5.4.** Computing the $k$th power of a $d$-regular constraint graph is computable by an NC algorithm if both $k$ and $d$ are in $O(1)$.

The assignment testing composition [8] Definition 5.1] consists of two parts. In the first part, each constraint is transformed into a Boolean circuit of constant size. In the second part, each circuit constructed in this way is provided as input to a computable assignment tester function (which we know exists [8] Theorem 5.1]), and the output graph is the union of the output of all the assignment testers. Since the size of the input to the assignment tester is constant, the assignment tester need only be computable. Hence, each constraint can be processed this way, in parallel, in constant time with respect to the size of the input graph.

**Lemma 5.5.** The assignment testing composition is computable by an NC algorithm.

Since the preprocessing function, constraint graph powering, and assignment testing composition are all computable by an NC algorithm, we conclude the following.

**Lemma 5.6.** There is a positive integer $q$ such that $\frac{1}{2}$-gap $q$-CSP is hard for NP under NC many-one reductions.

**Theorem 3.4** follows immediately from **Lemma 5.1** and **Lemma 5.6**.

**References**


